

# **Zusammenfassung DL4G**

## Deep Learning for Games

Maurin D. Thalmann

18. Januar 2020

# Inhaltsverzeichnis

<b>1</b>	<b>Sequential Games with perfect information</b>	<b>2</b>
1.1	Finite Sequential Games . . . . .	2
1.2	Complexity Factors in Game Analysis . . . . .	2
1.3	Illustration of State Space Complexity . . . . .	2
1.4	Extensive Form Representation . . . . .	2
1.5	Game Tree Analysis - Backward Induction . . . . .	3
1.6	Reasoning about Finite Sequential Games . . . . .	3
1.7	Zero-Sum Games . . . . .	4
1.8	Minimax Algorithm . . . . .	4
1.9	Search Tree Pruning . . . . .	5
1.10	Illustrations for Alpha-Beta Pruning . . . . .	5
<b>2</b>	<b>Monte Carlo Search Tree</b>	<b>6</b>
<b>3</b>	<b>Information Sets</b>	<b>6</b>
<b>4</b>	<b>Supervised Machine Learning</b>	<b>6</b>
<b>5</b>	<b>Neuronal Networks</b>	<b>6</b>
<b>6</b>	<b>Deep Neuronal Networks</b>	<b>6</b>
<b>7</b>	<b>Convolutional Neuronal Networks</b>	<b>6</b>

# 1 Sequential Games with perfect information

## 1.1 Finite Sequential Games

- Eine endliches Set an **Spielern**, jeder mit einem endlichen Set an möglichen **Aktionen**
- Spieler wählen ihre Aktionen **sequenziell** (einer nach dem anderen, in Zügen)
- Eine endliche Anzahl an **Zügen** wird gespielt
- Spätere Spieler **beobachten** die Züge der früheren Spieler (Perfect Recall)
- Eine **Strategie** sagt dem Spieler, welche Aktion er in seinem Zug spielen soll
- Ein **Strategieprofil** ist eine gewählte Strategie eines jeden Spielers
- Ein **Utility** oder **Payoff Function** bestimmt den Ausgang jedes Aktionprofils

## 1.2 Complexity Factors in Game Analysis

1. Anzahl Spieler
  - Spiele mit 4 Spielern sind schwieriger zu analysieren als solche mit 2 Spielern
2. Grösse des Suchraums
  - Bestimmt durch Anzahl gespielte Züge und Anzahl Aktionen für jeden Spieler
3. Kompetitive Spiele vs. Kooperative Spiele
  - Kompetitive Spiele involvieren Spieler mit komplett gegensätzlichen Interessen
4. Stochastische Spiele vs. Deterministische Spiele
  - Stochastische Spiele beinhalten Zufälle, bspw. Verteilung der Karten, Würfel rollen
5. Perfekte vs. imperfekte Informationsspiele
  - Imperfekte Information heisst das Spiel ist nur teilweise überwachbar, bspw. kennen wir nicht die Karten eines gegnerischen Spielers beim Poker oder Jass

## 1.3 Illustration of State Space Complexity

Game	State Space (as log to base 10; $10^x$ )
Tic-Tac-Toe	3
Connect-4	13
Backgammon	20
Chess	47
Go 19x19	170

- State Space beschreibt die Anzahl erlaubter Boardpositionen
- Schach / Chess hat  $10^{47}$  verschiedene Boards, Go hat  $10^{170}$  verschiedene Boards
- Zum Vergleich: Geschätzt sind im Universum  $10^{80}$  Atome

## 1.4 Extensive Form Representation

- Sequenzielle Spiele können (im Prinzip) als Spielbäume repräsentiert werden
- Knoten sind Spielzustände/Positionen und Kanten sind Aktionen/Bewegungen
- Blätter (Leaves) bestimmen den Payoff

## 1.5 Game Tree Analysis - Backward Induction

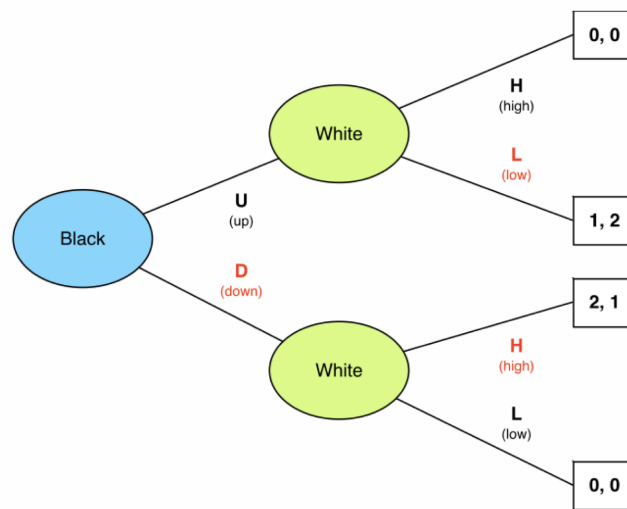


Abbildung 1: Backward Induction am Beispiel eines simplen Spielbaums

- Backward Induction ist der Lösungsalgorithmus für endliche, sequenzielle Spiele
- Im Beispiel oben hat Black den First-Mover Vorteil.
- Wenn beide Spieler perfekt spielen, endet das Spiel mit den Payoffs (2,1) (2 für Black, 1 für White)
- Solche Spiele immer rückwärts analysieren!

## 1.6 Reasoning about Finite Sequential Games

- Eine **ultra-schwache Lösung** beweist ob der erste Spieler aus der Initialposition gewinnen, verlieren oder unentschieden machen wir, in Annahme eines perfekten Spiels des Gegners  
*Im Beispiel: Schwarz kann einen Gewinn forcieren und hat demnach den First-Mover Vorteil*
- Eine **schwache Lösung** bietet einen Algorithmus welcher ein komplettes Spiel an perfekten Zügen aus der Initialposition offenbart, in Annahme eines perfekten Spiels des Gegners  
*Schwache Lösung für das Spiel: Black spielt U, White spielt L, Black spielt D*
- Eine **starke Lösung** bietet einen Algorithmus, welcher perfekte Züge aus jeder Position produzieren kann, auch wenn vorher von irgendeinem Spieler Fehler gemacht wurden.

*Starke Lösung für dieses Spiel:*

- Algorithmus für White: wenn Black U spielt → spiel L; wenn Black D spielt → spiel H
- Algorithmus für Black: Spiel D im ersten Zug; wenn White L im oberen Knoten spielt, spiel D

**A Strong Solution to Nim** Algorithmus für einen perfekten Nim Bot:

1. Wenn nur ein Haufen übrig bleibt  
→ Nimm alle Objekte des Haufens und hol den Preis
2. Wenn zwei Haufen mit unterschiedlicher Anzahl Objekte übrig bleiben  
→ Nimm Objekte vom grösseren Haufen und mache beide gleich gross
3. Wenn zwei Haufen diesselbe Anzahl Objekte haben  
→ Egal was du tust, du hast verloren (in Annahme eines perfekten Spiels des Gegners)

## 1.7 Zero-Sum Games

- Ein Spiel ist Zero-Sum, wenn der totale Gewinn des Siegers gleich dem totalen Verlust des Verlierers ist
  - Einen Kuchen zu schneiden ist zero-sum bspw. wenn ich ein Stück esse, ist es für dich verloren
  - Brettspiele sind zero-sum bspw. wenn der Sieger +1 erhält und der Verlierer -1
- $u_1$  und  $u_2$  umschreiben die Utility-Funktion von Spieler 1 und 2, somit ist für jedes Strategiepaar  $s_1$  und  $s_2$  von Spieler 1 und 2 und es gilt  $u_1(s_1, s_2) + u_2(s_1, s_2) = 0$ , dann lässt sich sagen:

$$u_1(s_1, s_2) = -u_2(s_1, s_2)$$

### Backward Induction & Minimax

- Backward Induction ist die Lösungsstrategie für endliche Spiele mit perfekter Information
- Eine einzelne Durchführung von Backward Induction aus einem Startzustand offenbart eine schwache Lösung. Wenn Backward Induction dynamisch (während des Spiels) aus jedem Zustand ausgeführt werden kann, erhalten wir eine starke Lösung.
- Wenn das Spiel zusätzlich zero-sum ist, kann Backward Induction mit dem Minimax Algorithmus implementiert werden. Minimax wird oft für Zwei-Spieler-Spiele definiert, ist aber auch für mehr Spieler erweiterbar.
- Minimax erlaubt effizientes Pruning („Ausästen“) und nahtlose Integration von Heuristiken
- Hinweis: Backward Induction kann auch für Spiele genutzt werden, die nicht zero-sum sind und kompliziertere Payoffs enthalten als eine einzelne Zahl

## 1.8 Minimax Algorithm

- 1928 von John von Neumann erfunden
- Beide Spieler wollen ihre respektiven Payoffs maximieren
- Weil das Spiel zero-sum ist, ist mein Gewinn = Verlust des Gegners
- Anstatt nach eigenem Gewinn zu maximieren, kann der Gewinn des Gegners minimiert werden
- Im Spielbaum kann folgendermassen vorgegangen werden:
  - Gehört der Knoten mir, wähle die Aktion welche den Payoff maximiert
  - Gehört der Knoten dem Gegner, wähle die Aktion welche den Payoff minimiert

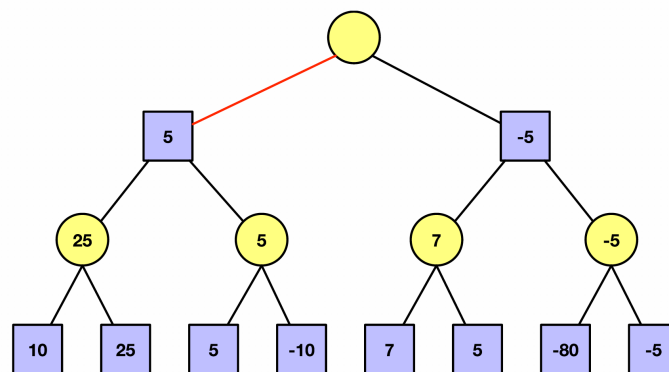


Abbildung 2: Lösungsweg eines Spielbaums mithilfe des Minimax-Algorithmus

### Programming a Minimax Bot

- Minimax implementiert Backward Induction für zero-sum Spiele. Dank der vereinfachenden zero-sum Eigenschaft können einige Tricks angewandt werden, um einen heuristischen Algorithmus zu erhalten:
  - Minimax nur bis zu einer limitierten Tiefe, bspw. 5 Runden vorausschauen und dann stoppen. Wie tief man gehen kann ist abhängig von den Spielregeln (Anzahl mögliche Züge), Effizienz der Implementation und Rechenleistung.

- Da die Tiefe limitiert ist, werden die Blattknoten nicht erreicht und wir kennen die echten Payoffs nicht. Wir müssen eine Heuristik erfinden, um die Situation abzuschätzen an welcher wir stoppen. Je besser die Heuristik, desto besser der Bot.

```

function minimax(node, depth, maximizingPlayer)
  if depth = 0 or node is a terminal node
    return the heuristic value of node
  if maximizingPlayer
    bestValue := -∞
    for each child of node
      val := minimax(child, depth - 1, FALSE)
      bestValue := max(bestValue, val)
    return bestValue
  else
    bestValue := +∞
    for each child of node
      val := minimax(child, depth - 1, TRUE)
      bestValue := min(bestValue, val)
    return bestValue

```

Abbildung 3: Pseudocode eines Minimax-Algorithmus mit limitierter Tiefe und Heuristik

## 1.9 Search Tree Pruning

- Es müssen nicht alle Knoten abgelaufen werden, um die optimale Strategie zu finden.
- Wir „stutzen“ (prunen) Sub-Bäume, welche keine bessere Lösung beinhalten können und demnach nicht besucht werden müssen.
- Dazu enthält der Algorithmus zwei Parameter:  
(Vorfahren sind alle Knoten auf dem Weg zwischen dem aktuellen und dem Root-Knoten)
  - $\alpha$  ist der höchste Wert aller MAX-Vorfahren eines MIN Knoten
  - $\beta$  ist der tiefste Wert aller MIN-Vorfahren eines MAX Knoten
- Der Algorithmus Alpha-Beta Pruning aktualisiert diese beiden Parameter im Minimax-Prozess und schneidet nicht besuchte Sub-Bäume ab, sobald er weiss dass die Werte aus diesem Sub-Baum den Wert  $\alpha$  nicht überbieten oder  $\beta$  nicht unterbieten können.

### Alpha-Beta Pruning Rules

- Alpha ( $\alpha$ ) ist der minimale Score, welcher dem maximierenden Spieler versichert werden kann
- Beta ( $\beta$ ) ist der maximale Score, welcher dem minimierenden Spieler versichert werden kann
- Daraus lassen sich folgende beiden Regeln schliessen:
  - **Regel 1:** Schneide ab, sobald der aktuelle Wert eines MIN Knoten kleiner ist als  $\alpha$
  - **Regel 2:** Schneide ab, sobald der aktuelle Wert eines MAX Knoten grösser ist als  $\beta$

## 1.10 Illustrations for Alpha-Beta Pruning

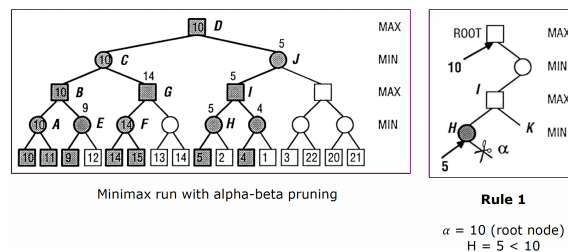


Abbildung 4: Illustration der Durchführung der 1. Regel des Alpha-Beta Pruning

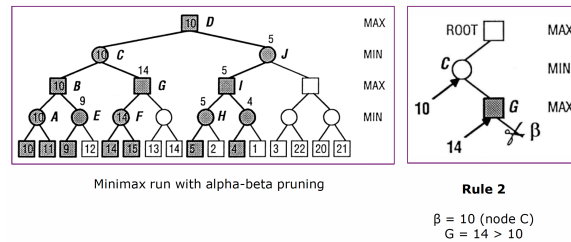


Abbildung 5: Illustration der Durchführung der 2. Regel des Alpha-Beta Pruning

```
function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        for each child of node
             $\alpha := \max(\alpha, \text{alphabeta}(\text{child}, \text{depth} - 1, \alpha, \beta, \text{FALSE}))$ 
            if  $\beta \leq \alpha$ 
                break (*  $\beta$  cut-off *)
        return  $\alpha$ 
    else
        for each child of node
             $\beta := \min(\beta, \text{alphabeta}(\text{child}, \text{depth} - 1, \alpha, \beta, \text{TRUE}))$ 
            if  $\beta \leq \alpha$ 
                break (*  $\alpha$  cut-off *)
        return  $\beta$ 
```

Abbildung 6: Pseudocode eines Minimax mit limitierter Tiefe mithilfe von Alpha-Beta Pruning

### Speed-Up of Alpha-Beta Pruning

- In einem Spielbaum mit Tiefe  $m$  mit  $b$  möglichen Aktionen bei jedem Knoten ist die Zeitkomplexität des Minimax  $O(b^m)$  bzw.. es gibt  $b^m$  Blattknoten
- Im Idealfall benötigt Alpha-Beta Pruning nur  $O(b^{m/2}) = O((\sqrt{b})^m)$ . Dies korrespondiert zu einer Reduzierung des Branching-Faktors von  $b$  zu  $\sqrt{b}$ , bspw. bei Schach bedeutet dies 6 mögliche Aktionen bei jedem Knoten (anstelle von 35)
- Um diesen maximalen Speed-Up zu erreichen, müssen die verschiedenen States in gescheiter Anordnung erforscht werden, was jedoch problemspezifisch ist.

## 2 Monte Carlo Search Tree

## 3 Information Sets

## 4 Supervised Machine Learning

## 5 Neuronal Networks

## 6 Deep Neuronal Networks

## 7 Convolutional Neuronal Networks